MC 910 2s 2010

Descrição do Projeto

Descrição Geral

O projeto consiste na implementação em Java de um compilador para uma linguagem de programação simples, gerando código para uma máquina virtual. O compilador deve ser organizado nos seguintes módulos:

- Análise Sintática
- Análise Semântica
- Geração de Código

Análise Sintática

O analisador sintático deverá ser construído a partir do CUP, que é um gerador de analisadores sintáticos baseado no método LALR. O pacote do CUP pode ser baixado em http://www2.cs.tum.edu/projects/cup/.

O manual de utilização pode está disponível em http://www2.cs.tum.edu/projects/cup/manual.html.

Para análise léxica poderá ser utilizado o JFlex, que é um gerador de analisadores léxicos. O pacote JFlex está disponível em http://jflex.de/ e o respectivo manual de usuário em http://jflex.de/manual.html.

A documentação do CUP descreve como integrar um *parser* gerado pelo CUP com um *scanner* gerado pelo JFlex e disponibiliza exemplos dessa integração em http://www2.cs.tum.edu/projects/cup/demo.zip e http://www2.cs.tum.edu/projects/cup/minimal.tar.gz (os comentários estão em alemão mas o código é de fácil compreensão).

O resultado da análise sintática dos programas sintaticamente corretos deverá ser uma representação intermediária baseada numa hierarquia de classes disponibilizada na página do curso. Essa representação intermediária, mantida em memória, será usada pelos demais módulos do compilador. No casso de programas com erros sintáticos, o compilador deverá fornecer um diagnóstico de erros que permita a correção dos mesmos pelo usuário do compilador.

Análise Semântica

A análise semântica será feita com base na representação intermediária construída pelo analisador sintático. A análise semântica consiste em verificar se o uso de cada 'elemento' do programa é consistente com a sua definição, no mesmo programa. Exemplos (não exaustivos) de situações que devem ser verificadas:

- Variáveis: para ser usada no programa uma variável deve ser previamente declarada.
- Expressões: em cada operação, os tipos dos operandos devem ser compatíveis com a operação.
- Comando condicional:
    - a expressão condicional deve ser verificada e o seu resultado deve ser do tipo 'boolean'

    - os comandos correspondentes aos casos ‘true’ e ‘false’ devem ser verificados.
- Comando de atribuição:
    - O tipo da variável deve ser ‘compatível para atribuição’ com o tipo do resultado da expressão.

O resultado da análise semântica será uma tabela de símbolos em memória, no caso dos programas ‘semanticamente corretos’. No caso de programas com erros de semântica, deverá ser emitida uma listagem com os erros encontrados. O objetivo dessa listagem é oferecer informações ao usuário do compilador que permitam a correção dos erros.

A tabela de símbolos será construída com base numa hierarquia de classes disponibilizada na página do curso. Essa tabela deverá ser mantida em memória para uso pelo gerador de código.

Geração de Código

O compilador deverá gerar código para uma máquina virtual cuja definição será disponibilizada na página do curso. A geração de código será feita a partir da representação intermediária gerada pelo analisador sintático e da tabela de símbolos resultante da análise semântica.

Equipes

O projeto deverá ser feito por equipes de até 2 pessoas.

Cronograma

- Análise sintática: entrega até 26/10.
- Análise semântica: entrega até 23/11
- Versão final: apresentação até 30/11

Definição da Linguagem

A linguagem que será objeto do projeto de curso, será chamada de L910, é baseada em C e Pascal e. A descrição a seguir é propositadamente informal, já que a formalização tanto da sintaxe como da semântica fazem parte do projeto. No texto a seguir as seguintes convenções são usadas:

- Texto entre “<” e “>” correspondem a elementos da linguagem descritos em alguma parte do texto.
- O texto entre “[“ e “]” se refere a uma construção opcional da linguagem.
- As partes em ´texto normal’ descrevem ao texto tal como se espera que apareçam nos programas escritos na linguagem (ou seja, se referem a símbolos terminais da gramática).

1. Tipos primitivos

`int`
`string`
`boolean`

2. Elementos léxicos

Identificadores:

Iniciados por letra, seguidos de letra, dígito ou '_'.

Palavras Reservadas:

```
int, boolean, void, string, true, false, if, else, while, ref,
read, readln, print, println, inc, dec, return
```

Constantes

- do tipo int: expressa como número inteiro em decimal.
- do tipo boolean: valores possíveis **true** ou **false**
- do tipo string: seqüência de caracteres entre aspas duplas
- tupla: sequência de constantes entre ´{´ e ´}´ e separadas por vírgula.

Operadores e delimitadores

- Os operadores são descritos adiante neste documento.
- Caracteres de tabulação, mudança de linha e comentários tem o mesmo significado que um espaço em branco.
- Uma seqüência de espaços em branco tem o mesmo significado que um único espaço em branco (ou seja, podem ser usados livremente para facilitar a legibilidade do programa fonte).

Nos identificadores e palavras reservadas, caracteres em maiúsculo são diferentes dos correspondentes em minúsculo.

3. Comentários

Comentários em L910 seguem as mesmas convenções de C:

```
/* isto é um comentário */
// isto é um comentário 'de linha'
/*    ====================
      um comentário pode
      ocupar diversas linhas
      do código e tem o mesmo
      'significado léxico' que um
      espaço em branco

      ====================
 */
```

4. Declaração de variáveis:

A declaração de variáveis segue o segunte padrão:

```
<tipo> <lista_de_variáveis> ';'
```

Onde <tipo> se refere a um tipo primitivo um e <lista_de_variáveis> se refere a uma seqüência de uma ou mais definições de variáveis separadas por vírgula.

Uma definição de variável na <lista_de_variáveis> é definida como

```
<identificador> [ <índice> ] [ '=' <valor_inicial> ]
```

Exemplos:

```
int a;
int x = 1, y, z = 3;
```

O valor inicial de uma variável é uma constante.

5. Vetores

Vetores são definidos através do uso de um <índice> na declaração de uma variável. Um <índice> é definido como

```
'[' <constante_inteira> ']'
```

onde <constante_inteira> define o tamanho do vetor. Ao se usar os elementos de um vetor numa expressão, os índices irão variar de 0 a tamanho-1.

Exemplos:

```
int a[10], x, b[10], c[10][10],y;
```

(nesse exemplo a e b são vetores com 10 elementos, c é um vetor de vetores, x e y são variáveis simples).

6. Tuplas

Uma tupla é uma seqüência de constantes separadas por vírgula e entre ´{´ e ´}´. Exemplos de tuplas:

```
{ 1, 2, 3 }
{ "um", "dois", "três" }
{ {1,2}, {3,4}, {4,5} }
```

Ao se atribuir uma tupla a um vetor, o número de elementos da tupla deve ser igual ao tamanho definido para o vetor, e a associação é feita elemento a elemento, mantendo-se no vetor a mesma seqüência dos elementos na tupla.

7. Strings

Constantes do tipo string serão escritos como uma seqüência de caracteres entre aspas duplas.

8. Exemplos de Declaração de variáveis:

```
int a,b=1, c;
boolean b1 = false;
char nome[30];
int idade,
int x;
int y;
```

```
boolean visible;

string d[7] = {"seg","ter","qua","qui","sex","sab","dom"};
```

9. Operadores aritméticos

Os operadores aritméticos em L910 são: '+', '-', '*', '/' e '%' (aplicáveis a operandos do tipo int). A sua precedência e semântica são equivalentes aos correspondentes em C e Pascal. O operador '-' pode ser utilizado como operador unário e nesse caso terá precedência maior que os demais operadores aritméticos.
Operador ´+´ poderá ser utilizado para somar valores inteiros e para a concatenação de strings. Esse operador pode ser usado em expressões mistas em que um dos operandos é do tipo string e nesse caso o resultado será do tipo string (a conversão do outro operando para string é implícita).
Exemplos:

- -10*30 será calculado como (-10)*30.
- 10 + "dez" terá como resultado "10dez"

10. Operadores Relacionais

Os operadores relacionais são '==', '!=', '>', '<', '>', '>=' e '<='. São aplicáveis a operandos de mesmo tipo e têm como resultado um valor do tipo boolean. Operadores relacionais têm precedência menor que os operadores aritméticos.

11. Operadores Lógicos

Os operadores lógicos são '||', '&&' e '!'. Os dois primeiros são operadores diádicos, sendo aplicáveis a operadores do tipo boolean, têm como resultado um valor também do tipo boolean e têm precedência menor que os operadores relacionais. O operador '!' (not) é um operador unário, tendo como operando do tipo boolean, retornando um valor de mesmo tipo. Este operador têm precedência maior que os demais operadores lógicos.
Exemplo: !a && b é calculado como (!a) && b.

12. Comando de Atribuição

O comando de atribuição tem a forma:

```
<variável> '=' <expressão> ';'
```

Onde variável pode ser

- um identificador representando uma variável simples
- um elemento de um vetor na forma <identificador>'['<índice>']', onde <índice> é uma expressão inteira cujo valor deve estar dentro dos limites definido para os índices do vetor.

Os tipos da variável e da expressão devem ser compatíveis para atribuição, o que em L910 significa que os tipos devem ser iguais ('char', 'boolean' e 'int', em L910 são tipos diferentes, ao contrário de C).

Exemplos: **`a = b+(c+1);  x = y; y  = 0;`**

13. Comando Condicional

O comando condicional tem a forma:

`'if ' '(' <expressão> ')' <comando> [ 'else' <comando> ]`

onde <expressão> deve retornar um valor do tipo boolean e <comando> pode ser qualquer comando da linguagem, inclusive um comando condicional. No de comandos condicionais encadeados onde uma parte 'else' possa ser associada a mais de um comando, esta deverá ser associada ao comando condicional mais interno. Exemplo:

`if (a > b) if(a > c) m = a else m = c;`

Neste exemplo, a parte 'else' é associada a 'if(a > c) ...'.

14. Comando composto

O comando composto é definido como

`'{' <declarações de variáveis> <seqüência_de_comandos> '}'`

onde <declarações de variáveis> é uma sequência zero ou mais declarações de variáveis e <seqüência_de_comandos> é uma sequência de zero ou mais comandos da linguagem. Em tempo de execução, os comandos são executados na mesma ordem em que aparecem na sequência que constitui o comando composto.

15. Comandos repetitivos

O comando repetitivo pode ter a seguinte forma:

`'while' '(' <expressão> '>')  <comando>`

16. Declaração de Função

Uma função é declarada da forma

`<tipo> <identificador> '(' <lista_parâmetros> ')' <corpo da função>`

onde

- <tipo> se refere a um dos tipos primitivos da linguagem ou 'void' (neste caso, indicando que a função não retorna um valor que possa ser usado numa expressão.
- <identificador> : é o identificador que define o nome da função.
- <lista_de_parâmetros>: corresponde à definição dos parâmetros formais da função. Os parâmetro são separados por vírgula.

- <corpo da função> : é um comando composto.

A <lista_de_parâmetros> é uma seqüência de zero ou mais definições de parâmetro separadas por vírgula. Uma definição de parâmetro tem a forma

['ref'] <tipo> <identificador>

O <corpo_da função> pode conter qualquer comando da linguagem e também o comando 'return' que é definido como

'return' [<expressão>] ';'

A execução deste comando causa o retorno ao ponto de onde foi feita a chamada à função. Se a função não retorna um valor ('void'), o comando 'return' não deve conter <expressão>. Caso retorne um valor de um tipo primitivo, <expressão> é necessária e seu valor deve ser compatível para atribuição com o tipo da função. O comando 'return' pode ocorrer mais de uma vez no corpo da função.
Exemplo:

```
int mdc(int a, int b){
          if(a == b) return a;
          if(b > a) return mdc(b,a);
          return mdc(a-b,b);
}
```

17. Chamada de função

A chamada a uma função tem a forma

<identificador> '(' <lista_de_valores> ')'

onde

- <identificador> corresponde ao identificador usado na declaração da função para definir o seu nome.
- <lista_de_valores> corresponde a uma lista de expressões separadas por vírgula representando os valores a serem associados aos parâmetros da função. O número de valores deve ser igual ao número de parâmetros da função. Os tipos dos valores devem ser compatíveis para atribuição com os tipos dos parâmetros.

Uma chamada de função pode ser usada como um comando da linguagem, na forma

<identificador> '(' <lista_de_valores> ')' ';'

Qualquer função definida num programa ou pré-definida pode ser chamada desta forma. Este comando não devolve um valor (caso a função retorne um valor, este é desprezado na execução do comando).

Uma chamada de função pode ainda ser usada como operando numa expressão. Neste caso, o tipo do valor de retorno deve ser compatível com a operação na qual a chamada de função é utilizada.

18. Passagem de parâmetros

A linguagem L910 segue as convenções de Pascal para a passagem de parâmetros. Nessa linha, são dois os modos de passagem de parâmetro:

- passagem por valor: cada valor de cada 'parâmetro de chamada' é calculado e passado à função como o valor inicial do parâmetro correspondente. Durante a execução da função um parâmetro passado por valor é tratado como se fosse uma variável local à função. Eventuais alterações no valor do parâmetro são locais à função e não afetam a função chamadora.
- Passagem por referência: o parâmetro de chamada deve ser uma variável e o valor passado à função é na verdade uma referência à variável da função chamadora. Caso o parâmetro seja alterado durante a execução da função, essa alteração é feita diretamente na variável da função chamadora.

Exemplo:

```
/** a e b são parâmetros passados por valor **/
void nao_troca(int a, int b){ int t = a; a = b; b = t; }

/** a e b são parâmetros passados por referência **/
void troca(ref int a, ref int b) { int t = a; a = b; b = t; }
```

19. Estrutura de programa e início de execução

Um programa em L910 é uma seqüência de declarações de variáveis seguida de uma sequência de declarações de funções. Uma vez declaradas, variáveis e funções podem ser usadas pelas funções subseqüentes. A execução do programa se inicia a partir da função 'main', cujo tipo e lista de parâmetros podem variar.

20. Visibilidade de nomes

Em L910 variáveis e funções só podem ser utilizadas após a sua declaração. Uma função pode conter declarações de variáveis (variáveis locais). Um comando composto também pode conter declarações de variáveis. Variáveis declaradas numa função só são 'visíveis' nessa função. Variáveis declaradas num comando composto só são visíveis nesse comando. Os nomes de variáveis ou funções devem ser únicos no escopo (programa, função ou comando) em que são declarados. Escopos diferentes podem ter nomes repetidos, e nesse caso, a declaração mais interna 'esconde' eventuais declarações do mesmo nome.

21. Tempo de vida de variáveis

As variáveis declaradas no programa, normalmente chamadas de 'variáveis globais' são criadas antes do início da execução do programa e só são destruídas após a execução do mesmo. As variáveis locais a uma função são criadas antes do início da execução da mesma e destruídas antes do retorno ao ponto de onde foi feita a chamada à função. Uma função

pode ser recursiva e nesse caso cada ativação terá o seu próprio conjunto de variáveis locais.

22. Funções pré-definidas

As funções pré-definidas têm como principal objetivo aumentar a usabilidade da linguagem. A definição de algumas delas se baseia em sintaxe e semântica diferente daquelas utilizadas pelas funções definidas pelo programador.

Entrada e saída
As funções de entrada e saída usam exclusivamente a entrada padrão e a saída padrão como dispositivos.

```
void read(<lista_de_variáveis>)
```

lê uma lista de valores pela entrada padrão e os associa às variáveis da lista. A <lista_de_variáveis> é uma lista de variáveis separadas por vírgula. Cada <variável> dessa lista obedece à mesma definição feita para o comando de atribuição.

```
void readln(<lista_de_variáveis> )
```

Semelhante à função read, com a seguinte diferença: após a leitura dos valores, ignora os caracteres da entrada padrão até uma mudança de linha.

```
void print(<lista de valores>)
void println(<lista de valores>)
```

Escreve uma seqüência de valores pela saída padrão. A <lista_de_valores> é uma seqüência de expressões separadas por vírgula. A função println é semelhante a print, com a seguinte diferença: após a escrita da seqüência de valores, muda para a próxima linha da saída padrão.

Incremento e Decremento de Variável

```
int inc(<variável>)
char inc(<variável>)
boolean inc(<variável>)
```

Altera o valor da variável para o 'próximo valor' de acordo com a seqüência de valores definidas no tipo da variável e retorna o novo valor da variável.

Exemplos:

```
inc(10) == 11
inc('a') == 'b'
inc(false)==true
inc(true) == indefinido.
```

```
int dec(<variável>)
boolean dec(<variável>)
```

Altera o valor da variável para o 'valor anterior', de acordo com a seqüência de valores definidas no tipo e retorna o novo valor da variável.

Exemplos:

```
dec(10)==9
dec(true) == false
dec(false) == indefinido.
```

Conversão de Tipos(*type casting*)

Uma variável de um tipo primitivo pode ser convertida para outro tipo através da operação de conversão de tipos, da forma

```
<nome_do_tipo>(<variável>)
```

onde <nome_do_tipo> corresponde ao nome de um dos tipos primitivos int ou boolean.

23. Erros em tempo de execução (exceções)

Embora a linguagem faça a verificação rígida de tipos, algumas exceções podem ocorrer durante a execução de um programa. Ao ocorrer uma dessas exceções o programa deve ser abortado e as bibliotecas de apoio devem fornecer uma indicação quanto ao tipo de exceção ocorrida e o número da linha onde ocorreu. Exemplos de situações não verificáveis em tempo de compilação que geram exceções em tempo de execução:

- Valor fora do intervalo permitido para o tipo.
- Valor de índice fora do intervalo definido para o vetor.
- Valor inválido numa operação de leitura.

Programas Exemplo

Exemplo 1: MDC e MMC

```
/** calcula o mdc entre dois inteiros **/
int mdc(int a, int b){
      while (a != b) do if(a > b) a = a-b; else b = b-a;
      return a;
}

/** calcula o mmc de dois inteiros **/
int mmc(int a, int b) {
   return (a*b)/mdc(a,b);
}

void main(){
   int a,b;
   readln(a,b);
```

```
   println(
            "a:",a," b:",b," mdc(a,b):",
            mdc(a,b)," mmc(a,b):",mmc(a,b)
          );
}
```

Exemplo2: Verificar se um número é primo

```
/** verifica se um número é primo **/
boolean isPrime(int n){
  int k = 2;
  while(k*k < n){
    if((n % k ) == 0) return false;
    k = k+1;
  }
  return true;
}

/** função main usada p/ testes **/
void main()
{
  int n;
  while(true){
    print("n:"); readln(n);
    if(n <= 0) return;
    if(isPrime(n)) println(n," é primo");
    else println(n," não é primo");
  }
}
```

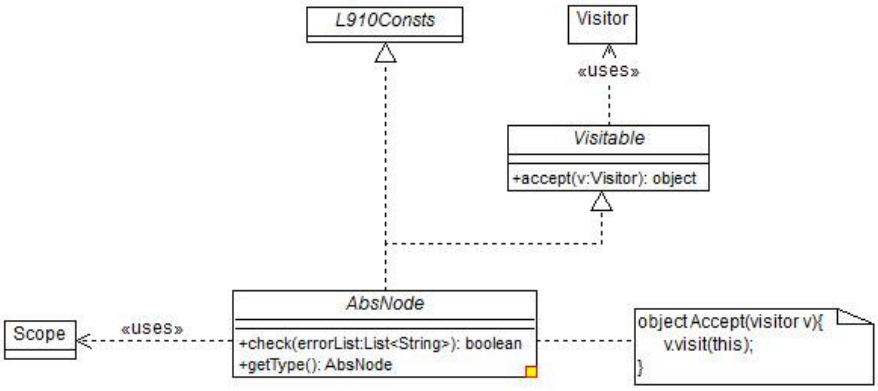
Exemplo 3 : bubblesort

```
/** bubblesort **/

int v[10]; // vetor a ser ordenado

/** troca o valor de duas variáveis **/
void troca(ref int a, ref int b){
    int t;
    t = a; a = b; b = t;
}

/** ordena o vetor v **/
void sort(){
   int k = 1, i;
   while (k > 0){
        k = 0;
        i = 0;
```

```
        while(i < 9)
           if(v[i] > v[i+1] { troca(v[i],v[i+1]); inc(k); }
        inc(i);
   }
}

void main(){
   int i;
   v = {99,9,88,8,77,7,66,6,55,5}
   sort();
   i = 0;
   while (i < 10) { print(v[i],"  "); inc(i); }
   println();
}
```

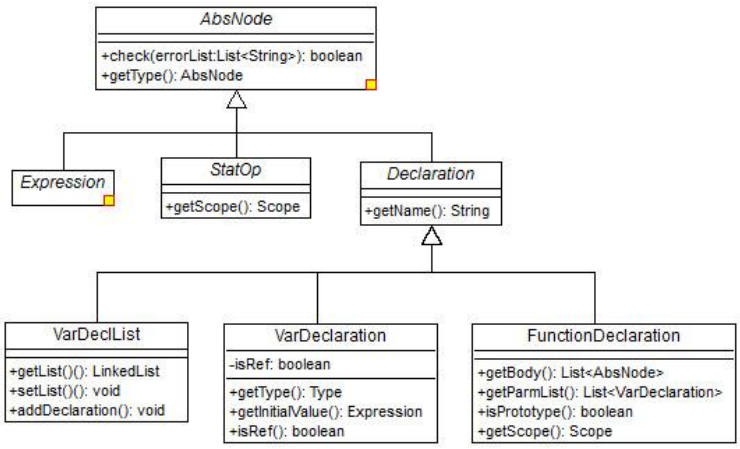
Representação Intermediária

A representação intermediária é uma estrutura de dados que tem por objetivo descrever a estrutura do programa numa forma que permita os vários tipos de percurso necessários às fases de verificação semântica e geração de código. Essa estrutura, em princípio, deve conter todas as informações importantes contidas no código fonte do programa sendo compilado. No caso deste projeto, a representação intermediária de programa será construída com base numa hierarquia de classes, cujo ´núcleo' é mostrado na figura a seguir.



Nessa figura a classe AbsNode é uma classe abstrata a partir da qual as demais classes da representação intermediária serão derivadas. Essa classe implementa as interfaces L910Consts e Visitable. A primeira define um conjunto de constantes que podem ser usadas no compilador e a segunda define a interface para a utilização do *design pattern* Visitor.

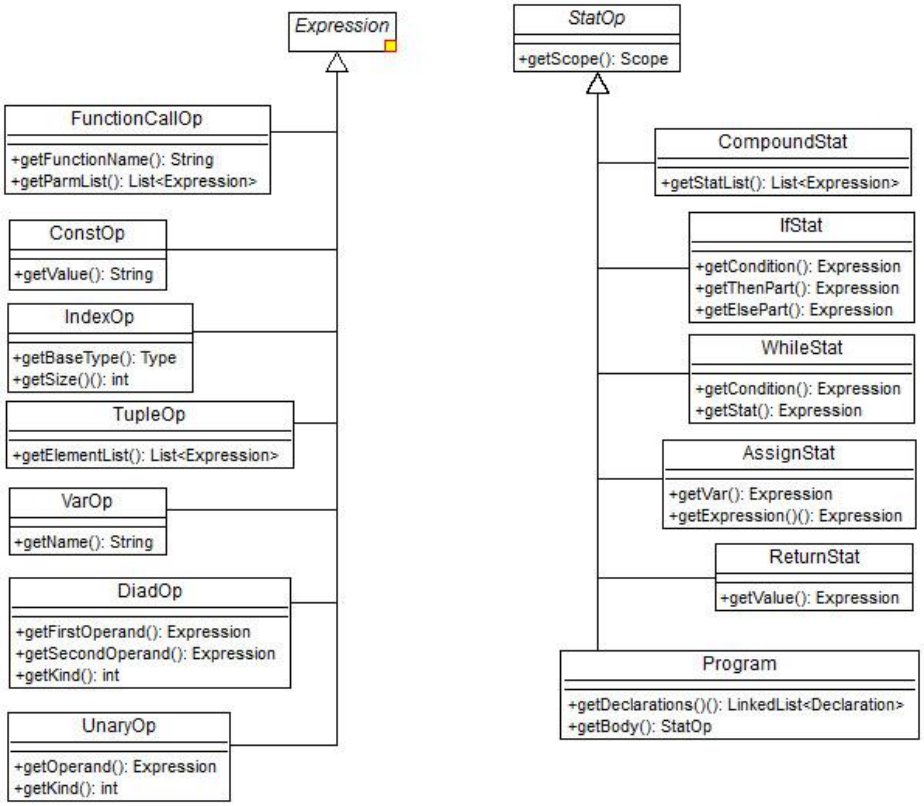
A figura a seguir mostra as classes derivadas de AbsNode:



Nessa figura

- Expression é uma classe abstrata que dá origem às classes usadas na representação das expressões que compõem o programa sendo compilado.
- StatOp é uma classe abstrata a partir da qual são derivadas as classes que descrevem os comandos do programa.
- Declaration é uma classe abstrata a partir da qual são criadas
    - VarDeclaration descreve as declarações de variáveis feitas no programa.
    - FunctionDeclaration descreve as declarações de funções no programa.

A classes Expression e StatOp por sua vez dão origem outras classes, mostradas a seguir:



Visitor

O *design pattern* Visitor deverá ser utilizado na construção das classes responsáveis pela verificação semântica e geração de código. Esse *pattern* permite a implementação dessas funcionalidades sem nenhuma alteração na representação intermediária.

A título de exemplo, o pacote que constitui a representação intermediária oferece uma classe chamada ‘SampleVisitor’ que gera, a partir da representação intermediária, um texto fonte equivalente ao programa representado pela mesma.